Discrete Applied Mathematics 2 (1980) 151–153

NOTE

AN O(m log D) ALGORITHM FOR SHORTEST PATHS

Pierre HANSEN

Feculté Universitaire Catholique de Mons, Belgium, and Institut d'Economie Scientifique et de Gestion, Lille, France.

Received 2 March 1979
Revised 14 December 1979

An implementation of Dykstra's shortest paths algorithm is proposed, which requires O(m log D) computations in worst case, where m denotes the number of arcs and D the length of the longest arc of the graphs considered. To this effect, a data structure called *binary counting tree* is introduced.

1. Introduction

Let $G=(X, U)$ denote a directed graph, with $n=|X|$ and $m=|U|$, to the arcs $(x_k, x_l)\in U$ of which are associated lengths d_{kl}. The problem of finding shortest paths between a vertex x_1 of G and all others has been much studied (see e.g. Christofides [1], Dreyfus [4], Denardo and Fox [2], Lawler [9]). When the $d_{kl}\geq 0$, Dykstra's [5] labelling algorithm applies:

(a) *Initialisation.* Set $\lambda_1=p_1=0$, $\lambda_j=\infty$ and $p_j=0$ for $j=2, 3, \ldots, n$, $T=\{1, 2, \ldots, n\}$.

(b) *Selection of the vertex with the smallest temporary label.* If $T=\emptyset$, end. Otherwise choose x_k such that $\lambda_k=\min\{\lambda_j \mid j\in T\}$. Set $T:=T\backslash\{k\}$.

(c) *Updating of labels.* If $\lambda_k=\infty$, end. Otherwise, for each x_l such that $(x_k, x_l)\in U$ and $\lambda_l>\lambda_k+d_{kl}$ set $\lambda_l=\lambda_k+d_{kl}$ and $p_l=k$. Go to (b).

The final λ_j are equal to the lengths of the shortest paths, these paths may be obtained recursively through the predecessor indices p_j. So stated, the algorithm appears as a *first-generation* one, i.e. as a sequence of mathematical rules, without discussion of the data structures needed for an efficient implementation. Filling in the details yields *second-generation* algorithms, with performance guarantees. An obvious implementation requires O(n^2) computations, while, as all arcs must be considered, at least O(m) computations are required. Step (a) and step (c) take O(n) and O(m) computations, respectively, so step (b) is the crucial one. For sparse graphs much less than O(n^2) computations are required. Using a *heap* to store the temporary labels yields an O(m log n) algorithm, c-*heaps* may be used

also (see Johnson [8], Johnson [7], Denardo and Fox [2]). When the d_{kl} are integers the temporary labels may be associated with the lines of a table where the indices of the so-labelled vertices are stored (i.e. if $\lambda_3 = 5$, the index 3 is stored in line 5, etc. see Dial [3], Loubal, according to Hitchner [6], Wagner [10]). Moreover, as at a current iteration the finite temporary labels differ by $D = \max(d_{kl} \mid (x_k, x_l) \in U))$ at most, a *rolling-over table* of length $D+1$ may be used (i.e. the first lines are re-used to store the indices of the vertices with labels $>D+1$, then $>2D+2$, etc.). Finally, such a table may be stored compactly with a vector of length $D+1$ containing the last index stored in each line, and two vectors of length n for a *double chaining* allowing to add or delete any index to or from a line. Then a *pseudo-polynomial algorithm* which requires $O(\max(nD, m))$ computations is obtained. Clearly, it is most adequate for small D. The purpose of the present note is to propose a modification of that implementation which is efficient even for large D.

2. Binary counting trees

Let p denote the smallest integer such that $2^p > D$. Let us associate to the table, assumed of length 2^p, in which the indices of the temporary labels are stored a *binary tree* of *height* p defined as follows: a leaf corresponds to each line, with a label equal to the number of indices of that line; there are no other leaves. A vertex of height $p-q$ for $q = 1, 2, \ldots, p$ corresponds to 2^q adjacent lines and has a right son and a left son corresponding to the 2^{q-1} first and 2^{q-1} last of these lines, respectively. The label of any vertex which is not a leaf is equal to the sum of the labels of its sons. Such a structure may be called *binary counting tree* (b.c.t.).

Finding the first non-empty line in the table can be done by applying the following rule recursively from the root of the b.c.t.: "If label (right son $(x)) \neq 0$, go to right son (x), otherwise go to left son (x)". If the table is a rolling-over one and k indices are stored in the upper, re-used part that rule must be modified as below to find the line containing the index of the vertex with smallest label: "If label (right son $(x)) > k$ go to right son (x), otherwise set $k := k$-label (right son (x)) and go to left son (x)". The next rule allows to find the root-leaf path in the b.c.t. to the tth line, after setting $q = 1$: "If $2^{p-q} \geq t$ go to right son (x), otherwise set $t := t - 2^{p-q}$ and go to left son (x); set $q := q + 1$".

When a b.c.t. is used, step (b) of Dykstra's algorithm is a straightforward application of the second rule and takes $O(p)$ i.e. $O(\log D)$ computations each time it is used. Deleting or adding the index of a vertex whose label has changed in step (c) requires finding the corresponding root-leaf path by the third rule, subtracting or adding one to the labels of its vertices and updating the double chaining. This takes $O(\log D)$ computations also, and as there are at most m label

modifications step (c) takes $O(m \log D)$ computations in all, as does the whole algorithm (provided $m \geq n$).

Moreover, *only the vertices of the b.c.t. with non-zero labels need be stored*, and at any current iteration there are less than np such vertices. To this effect four lists of length np must be used, the first for the labels, the second and third for pointers to the positions of the left sons and right sons, or when the vertices are leaves to the last indices in the corresponding lines, the fourth to store the indices of the empty spaces in the three first. Including space for the data, $2m+(4p+5)n$ memory spaces are needed, at most.

The method of the introduction, with rolling-over table and double-chaining, requires $2m+5n+D$ memories, i.e. more than the proposed method when $D>4n \log D$. For graphs which are paths from x_1 to x_n with $n-1$ arcs of length D, the first method requires $(n-1)D$ operations to scan the table for the first non-empty line while the second one requires $(n-1)\log D$ operations only for that purpose.

Finally, let us note that using a b.c.t. yields of course a $O(n \log D)$ algorithm for shortest paths in planar graphs, and that a similar $O(m \log D)$ algorithm for minimum spanning trees is easily obtained.

References

[1] N. Christofides, Graph Theory, An Algorithmic Approach (Academic Press, New York, 1975).
[2] E.V. Denardo and B.L. Fox, Shortest route methods: 1. Reaching, pruning and buckets; Operations Research 27 (1979) 161–186.
[3] R.B. Dial, Algorithm 360: Shortest path forest with topological ordering, Comm. Assoc. Comput. Mach. 12 (1969) 632–633.
[4] S.E. Dreyfus, An appraisal of some shortest-path algorithms, Operations Res. 17 (1969) 395–412.
[5] E.W. Dykstra, A note on two problems in connection with graphs, Numerische Mathematiek 1 (1959) 269–271.
[6] L.E. Hitchner, A comparative investigation of the computational efficiency of shortest path algorithms, Report ORC 68-25, University of California, Berkeley (1968).
[7] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, J. Assoc. Comput. Mach. 24 (1977) 1–13.
[8] E.L. Johnson, On shortest paths and sorting, in: Proceedings 1972 ACM National Conference (1972) 510–517.
[9] E.L. Lawler, Combinatorial Optimization, Networks and Matroids, (Holt, Rinehart and Whinston, New York, 1976).
[10] R.A. Wagner, A shortest path algorithm for edge-sparse graphs, J. Assoc. Comput. Mach. 23 (1976) 50–57.